Improving your ObjectARX® C++ Code
Mike Dickason – Autodesk, Inc.

CP5177: From memory leaks to performance bottlenecks and crashes, this class will present issues that are potentially causing problems with your ObjectARX C++ code. The class will show how to identify potential problems in your C++ code and recommend solutions. Class attendees are expected to have a working knowledge of ObjectARX C++. This is not a class for someone hoping to learn to program in C++.

Learning Objectives
At the end of this class, you will be able to:

- Identify potential coding styles in your ObjectARX C++ code which could lead to product stability issues
- Write better quality ObjectARX C++ code
- Identify potential performance problems with your existing ObjectARX C++ code
- Improve the quality of your existing ObjectARX C++ code

About the Speaker
Mike Dickason has a degree in Civil Engineering and has been working as a software developer at Autodesk for the last 17 years. He is currently working on the AutoCAD® product line and previously worked on both AutoCAD Mechanical Desktop and AutoCAD Mechanical.

mike.dickason@autodesk.com
Introduction
This session is going to present some common coding issues which impact ObjectARX® (ARX) applications. The focus will be on stability, memory leaks, and performance of your C++ based ARX code. The material presented here is based on the assumption you are already experienced with writing ARX C++ code and are looking for tips on how to improve the quality of your code. I won’t be discussing Mac development specifically, but the majority of the items presented here are applicable to both Windows and Mac platforms.

Stability
Based on internal development and the crash reports our customers submit to Autodesk, here are several common ARX coding issues which result in crashes.

1.  **Holding onto an AcDbObjectId after its database has been destroyed**  
   This is without question the number one cause of ARX crashes. An AcDbObjectId is only valid through the life of the AcDbDatabase. Once the AcDbDatabase has been destroyed, any attempt to use an AcDbObjectId from that database will result in a crash.

   ```cpp
   // Stale object id – bad example #1
   // m_PickFirstIdList is an AcDbObjectIdArray data member on a global reactor class
   void CustomDbReactor::objectErased(AcDbDatabase* pb,
                                       const AcDbObject* pObj,
                                       Adesk::Boolean erased)
   {
     if (pObj && pObj->isKindOf(MyCustomObject::desc())) {
       if (!m_PickFirstIdList.contains(pObj->objectId()))
         m_PickFirstIdList.append(pObj->objectId()); // Danger, Will Robinson!
     }
   }

   // Stale object id – bad example #2
   // m_PickFirstIdList is an AcDbObjectIdArray data member on a global reactor class
   void CustomDbReactor::doSomeThing()
   {
     if (m_PickFirstIdList.isEmpty())
       return;
     AcDbObject* pObj = NULL;
     // If the last id in the array is bad, the acdbOpenObject() call will crash
     if (Acad::eOk == acdbOpenObject(pObj, m_PickFirstIdList.last(), AcDb::kForRead) {  
       // Do something with pObj;
       pObj->close();
     }
   }
   ```
The first example will potentially crash with the m_PickFirstIdList.contains() call. Once the database is deleted, if m_PickFirstIdList has any ids in it from the database, the attempt to search the array for an object id will result crash when the AcDbObjectId::operator==() method is executed.

The second example will potentially crash with the acdbOpenObject() call for the same reason.

2. Lack of error checking
This one is actually applicable to programming in general and not just to ARX. The code you write needs to test return values and take appropriate action when a bad status is encountered. Unfortunately, the crash data we receive indicates that there is a definite lack of error checking happening in ARX applications.

   a. upgradeOpen() failures
      // create a new visual style – bad example
      AcDbDictionary *pVisualStyleDict = NULL;
      if (Acad::eOk == pDb->getVisualStyleDictionary(pVisualStyleDict, AcDb::kForRead)) {
         AcDbVisualStyle* pNewVisualStyle = new AcDbVisualStyle();
         pNewVisualStyle->setDescription(newVisualStyleName);
         pNewVisualStyle->setType(AcGiVisualStyle::kCustom);
      
         pVisualStyleDict->upgradeOpen();  // Danger, Will Robinson!
         pVisualStyleDict->setAt(newVisualStyleName, pNewVisualStyle, newVisualStyleId);
         pVisualStyleDict->downgradeOpen();
      
         pNewVisualStyle->close();
         pVisualStyleDict->close();
      }

While the example here deals with the visual style dictionary, the issue is the same regardless of the object type. There are multiple reasons that the upgradeOpen() call could fail and return something other than Acad::eOk. Somebody could already have the object open for read. The calling routine might have neglected to obtain a write lock on the document. Whatever the reason, because the object is still only open for read, the setAt() call will result in a fatal error since it is attempting to modify a read-only object.

The first attempt to address this might look something like the following:

   // create a new visual style
   AcDbDictionary *pVisualStyleDict = NULL;
   if (Acad::eOk == pDb->getVisualStyleDictionary(pVisualStyleDict, AcDb::kForRead)) {
      AcDbVisualStyle* pNewVisualStyle = new AcDbVisualStyle();
      pNewVisualStyle->setDescription(newVisualStyleName);
// set with the custom type
pNewVisualStyle->setType(AcGiVisualStyle::kCustom);

if (Acad::eOk == pVisualStyleDict->upgradeOpen()) {
    pVisualStyleDict->setAt(newVisualStyleName, pNewVisualStyle,
    newVisualStyleId);
    pVisualStyleDict->downgradeOpen();
}
pNewVisualStyle->close();
pVisualStyleDict->close();

Does anyone see a potential problem with this solution? (Hint: Memory leak)

b. locked layers
Locked layers come into play when you attempt to open an object for write.

    // Locked layer – bad coding example #1
    AcDbEntity* pEnt = NULL;
    acdbOpenObject(pEnt, entId, AcDb::kForWrite); // Danger, Will Robinson!
    pEnt->setLayer(newLayerId);
    pEnt->close();

    // Locked layer – bad coding example #2
    AcDbEntity* pEnt = NULL;
    if (Acad::eOk == acdbOpenObject(pEnt, entId, AcDb::kForWrite)) {
        pEnt->setLayer(newLayerId);
    }
    pEnt->close(); // Danger, Will Robinson!

    // Locked layer – bad coding example #3
    AcDbObjectPointer<AcDbEntity> pEnt(entId, AcDb::kForRead);
    if (Acad::eOk == pEnt.openStatus()) {
        pEnt->upgradeOpen(); // Danger, Will Robinson!
        pEnt->setLayer(newLayerId);
    }

The first example is bad because it fails to test the return status from the acdbOpenObject() call and assumes that the open will always succeed. If the entity is on a locked layer, the call will return a bad status and the subsequent attempt to use pEnt will result in a crash.
The second example is bad because the close() call it outside of the scope of the if() statement. So if the acdbOpenObject() returns a bad status the code will still crash when it attempts to use the NULL pEnt.

The third example is bad because it fails to check the return value from the upgradeOpen() call. If the entity is on a locked layer, the call will return a bad status and the subsequent attempt to use pEnt will result in a crash.

c. Attempting to open erased objects
Similar to the previous example with locked layers, attempting to open an erased object will result in a bad return status unless you specifically specify that you want to open erased objects.

    // Open erased object – bad coding example #1
    AcDbEntity* pEnt; // Danger, Will Robinson! Potentially uninitialized variable
    acdbOpenObject(pEnt, entId, AcDb::kForRead); // Danger, Will Robinson!
    AcDbObjectId layered = pEnt->layerId();
    pEnt->close();

If entId is the object id of an erased object, the acdbOpenObject() call will return Acad::eWasErased and pEnt will be a garbage pointer.

    // Open erased object – good coding example #1
    AcDbEntity* pEnt = NULL;
    AcDbObjectId layerId;
    if (Acad::eOk == acdbOpenObject(pEnt, entId, AcDb::kForRead, true)) {
        layerId = pEnt->layerId();
        pEnt->close();
    }

d. Identifying potential problem spots
The following regular expression can be used in DevStudio to search your code for locations where the return value from acdbOpenObject() isn’t being checked.

    ^\s*acdbOpenObject

3. Failing to close all opened objects
This is a subtle but nasty case since the ultimate failure point is likely nowhere near the actual problem location. Every call to open an object must be matched with a call to close the object.

    // Object open – bad example
    AcDbEntity* pEnt;
    acdbOpenAcDbEntity(pEnt, entId, AcDb::kForRead);
if (pEnt->layerId() != searchId) {
    continue; // Danger, Will Robinson! pEnt is left open for read
}

// Object open – good example
AcDbObjectPointer<AcDbEntity> pEnt(entId, AcDb::kForRead);
if (Acad::eOk != pEnt.openStatus())
    continue;
if (pEnt->layerId() != searchId) {
    continue; // The AcDbObjectPointer destructor will automatically close pEnt
}

If you aren’t already doing so, you should start using the AcDbObjectPointer template instead of the legacy acdbOpenObject(), acdbOpenAcDbEntity(), acdbOpenAcDbObject() APIs.

4. Holding objects open longer than necessary
It’s bad form to hold objects open for read or write longer than absolutely necessary. You should get into the habit of closing objects as soon as you are done using them. For example, say you need to make copies of all of the mleader objects in the current paperspace layout.

    // Holding objects open longer than necessary – bad example
    Acad::ErrorStatus es processPspaceMleaders()
    {
        AcDbDatabase* pDb = acdbHostApplicationServices()->workingDatabase();
        if (!pDb)
            return Acad::eInvalidInput;

        AcDbBlockTablePointer pBlockTable(pDb, AcDb::kForRead);
        if (Acad::eOk != pBlockTable.openStatus())
            return pBlockTable.openStatus();

        AcDbObjectId paperSpaceId;
        Acad::ErrorStatus es = pBlockTable->getAt(ACDB_PAPER_SPACE, paperSpaceId);
        if (Acad::eOk != es)
            return es;

        //
        // Iterate the block table record and process all of the mleaders
        //

        return es;
    }
With this example, once the object id of the paperspace block table record is obtained the AcDbBlockTable is no longer needed. But the code holds the block table open for read until the final return at which point the AcDbBlockTablePointer destructor closes the table. This can cause unexpected problems. With this example imagine the code makes copies of the mleaders and during the copying process one of the mleaders being copied needs to modify the block table. Because the sample code here already has the block table open for read, the mleader would be unable to modify the block table because it would be unable to open the table for write and the copy would fail.

Here's the correct version:

```cpp
// Holding objects open longer than necessary – good example
Acad::ErrorStatus es processPspaceMleaders()
{
    AcDbDatabase* pDb = acdbHostApplicationServices()->workingDatabase();
    if (!pDb)
        return Acad::eInvalidInput;

    AcDbBlockTablePointer pBlockTable(pDb, AcDb::kForRead);
    if (Acad::eOk != pBlockTable.openStatus())
        return pBlockTable.openStatus();

    AcDbObjectId paperSpaceId;
    Acad::ErrorStatus es = pBlockTable->getAt(ACDB_PAPER_SPACE, paperSpaceId);
    if (Acad::eOk != es)
        return es;

    // We are done with the block table, so close it now.
    pBlockTable.close();

    //
    // Iterate the block table record and process all of the mleaders
    //
    return es;
}
```

5. **Avoid accessing into an empty AcArray**

Given a class MyObject with a data member having the following declaration:

```cpp
    AcArray<AcDbEntity*> m_theArray;
```

Question: What do these three code snippets have in common?
// AcArray bad coding example #1
AcDbEntity* MyObject::getFirst()
{
    return m_theArray.first();
}

// AcArray bad coding example #2
AcDbEntity* MyObject::getLast()
{
    return m_theArray.last();
}

// AcArray bad coding example #3
AcDbEntity* MyObject::getIndex(int index)
{
    return m_theArray[index];
}

Answer: All three versions will result in a crash if m_theArray is empty. Before attempting to access the contents of an AcArray, you must ensure that the array is not empty by calling AcArray::isEmpty().

6. **CString Abuse**
Two common usages of the CString class can result in crashing your application.

   a. **CString::Format()**
   Here’s what the MSDN help says about this function:

   The call will fail if the string object itself is offered as a parameter to Format. For example, the following code:

   CString str = "Some Data";
   str.Format("%s%d", str, 123);  // Attention: str is also used in the parameter list.

   will cause unpredictable results.

   What it should say is that it will crash.

   Here are two different regular expressions you can use in DevStudio to try and locate potential types of this abuse:
b. **CString[] with a bad index**

Just like the example above with the AcArray, attempting to index into a CString with an invalid index will throw an exception. It used to be that the CString code trapped these attempts to index outside of the string and simply returned an empty string, but a couple releases back the class was changed to instead throw an exception.

```cpp
CString str = getApplicationPath();
// Check the last character for a backslash
    str += '/';
```

### 7. **CFile and CFileExceptions**

The majority of the CFile methods have the potential to throw a CFileException, which if uncaught will take down the application. For example:

```cpp
TCHAR* pFileName = _T("test.dat");
CFile::Remove(pFileName);
```

If the user doesn’t have the necessary privileges to delete the file, then a CFileException is thrown.

For a stable application, every call to a CFile method needs to be in a try/catch block similar to the following:

```cpp
//example for CFile::Remove
TCHAR* pFileName = _T("test.dat");
try
{
    CFile::Remove(pFileName);
}
catch (CFileException* pEx)
{
    // the attempt to delete the file failed. Now what?
    pEx->Delete(); // If you are done with the exception, don’t forget to delete it
    // or you will be leaking memory.
}```
8. *isA() vs isKindOf()*

It is important to understand the difference between *isA()* and *isKindOf()* when determining a class type. You should only use *isA()* when you are looking for a specific class type and do not want to consider derived classes. For example, say you have a specific reason to process only AcDbBlockReference instances and you need to skip any derived classes such as AcDbTable or AcDbMInsertBlock.

```cpp
// This handles the AcDbBlockReference and all derived objects
if (pEnt->isKindOf(AcDbBlockReference::desc())) {
    AcDbBlockReference* pBlkref = static_cast<AcDbBlockReference*>(pEnt);
    // do something here with pBlkref
}

// This handles ONLY AcDbBlockReference instances. All derived classes are ignored
if (pEnt->isA() == AcDbBlockReference::desc()) {
    AcDbBlockReference* pBlkref = static_cast<AcDbBlockReference*>(pEnt);
    // do something here with pBlkref
}
```

Any time you see code using *isA()* as a type check, the first thing you should ask yourself is whether or not the code is intended to work with derived objects.

9. **Don’t call delete on database resident objects**

Calling delete on an AcDbObject* will result an immediate crash if the object is database resident. If you have an object pointer and there’s a chance it’s a database resident object, then you need to test that before calling delete.

```cpp
// delete - bad example
void mySpecialExplode(const AcDbEntity* pEnt)
{
    AcDbVoidPtrArray newEnts;
    if (Acad::eOk == pEnt->explode(newEnts)) {
        // process the contents of newEnts here and add each entry to the database
        processTheClones(newEnts);
    }
    for (int index = 0; index < newEnts.length(); ++index)
        // Danger Will Robinson - deleting a database resident object
        delete static_cast<AcDbObject*>(newEnts[index]);
    return;
}
```
As long as there is no chance of processTheClones() leaving an entry in the array after it has been added to the database, the code as written should work without error. Otherwise, it needs to be rewritten as follows:

```c++
// delete object pointer – good example
void mySpecialExplode(const AcDbEntity* pEnt)
{
    AcDbVoidPtrArray newEnts;
    if (Acad::eOk == pEnt->explode(newEnts)) {
        // process the contents of newEnts here and add each entry to the database
    }
    for (int index = 0; index < newEnts.length(); ++index) {
        AcDbObject* pObj = newEnts[index];
        if (!pObj)
            continue;
        if (NULL != pObj->database())
            pObj->close();  // It has a database, we can't delete it
        else
            delete pObj;
    }
    return;
}
```

10. PDB generation and storage

While it won’t help prevent crashes, having your application properly configured to generate .PDB files can help you debug reported crashes. It is critical that your projects are properly configured to generate the .PDB files for both debug and release builds at the time. If you discover that you don’t have .PDB generation enabled, it is impossible to generate the .PDB files after the fact. For example, you release v1.0 of your product and then start getting customer reports of crashes. One of the customers sends you the .dmp file associated with the crash and when you load it into the debugger DevStudio informs you that no symbols could be found for the module. You investigate and find out that .PDB generation wasn’t enabled for your module. At this point the .dmp file is effectively worthless. Even if you rebuild the module with the exact same code that was used for the released module, the resulting .PDB file will not work with the .dmp file.

Unfortunately it is easy to improperly configure your project which results in no .PDB files being generated. There are two settings which have to be enabled in your project in order to generate .PDB files. The first setting is on the C/C++ General tab as indicated below:
The second setting is on the Linker Debugging tab as indicated below:
You need to make sure that both settings are set for each build configuration in your project.

11. **Using a symbol server**
Once you have your build setup to properly generate .PDB files with your binaries, you need to copy them into a symbol server. Symbols on the server should be maintained for as long as the build is valid. If it is an internal build that customers won’t see, you might want to keep the current build and two previous builds. If it is a build customers will get, then you need to keep the symbols for as long as you expect customers to be using the build.

Here is a Microsoft link on how to set up your symbol server:


Kean Walmsley has an excellent post on his blog detailing how to access the symbols for the AutoCAD 2012 release. That post can be found at the following URL:

Memory leaks
The following is a list of some of the more common memory leaks that show up in ARX applications. It by no means is meant to be comprehensive list of every possible way to leak memory.

1. **AcDbEntity::layer()**
The AcDbEntity::layer() method always returns an allocated string which the caller is responsible for freeing.

   Question: What do the following code samples have in common?

   ```cpp
   // Obtaining the entity layer – bad example #1
   pEntCopy->setLayer(pEnt->layer());
   
   // Obtaining the entity layer – bad example #2
   CString layerName(pEnt->layer());
   ```

   Answer: Both obtain the string for the name of the layer the entity, and both are leaking memory.

   To avoid the leaks each needs to be rewritten as:

   ```cpp
   // Obtaining the entity layer – good example #1
   TCHAR* pLyr = pEnt->layer();
   pEntCopy->setLayer(pLyr);
   acutDelString(pLyr);
   
   // Obtaining the entity layer – good example #2
   TCHAR* pLyr = pEnt->layer();
   CString layerName(pLyr);
   acutDelString(pLyr);
   ```

2. **AcDbSymbolTableRecord::getName()**
There are three implementations of getName() to obtain the name from a symbol table record. One of the three returns an allocated string which the caller is responsible for freeing.

   ```cpp
   // Retrieve the symbtol name record name – bad example #1
   CString getSymbolTableName(const AcDbSymbolTableRecord* pSym)
   {
       ACHAR* pName; // Danger, Will Robinson! Uninitialized pointer
       pSym->getName(pName); // Uh-oh, didn’t check the return value
       return CString(pName); // Uh-oh, we just leaked memory
   }
   ```

   To fix this leak you would need to change the code to the following:
// Retrieve the symbol name record name – good example #1
CString getSymbolTableName(const AcDbSymbolTableRecord* pSym)
{
    const ACHAR* pName = NULL;
pSym->getName(pName);
    return CString(pName);
}

3. **AcDbEntity::explode()**
This method populates an array with non-database-resident entity pointers which the caller is responsible to delete.

// Explode – bad example #1
void mySpecialExplode(const AcDbEntity* pEnt)
{
    AcDbVoidPtrArray newEnts;
    if (Acad::eOk == pEnt->explode(newEnts)) {
        // process the contents of newEnts here but don’t add them to the database
    }
    return; // Danger Will Robinson, we’re leaking memory
}

// Explode – bad example #2
void mySpecialExplode(const AcDbEntity* pEnt)
{
    AcDbVoidPtrArray newEnts;
    if (Acad::eOk == pEnt->explode(newEnts)) {
        // process the contents of newEnts here but don’t add them to the database
    }
    for (int index = 0; index < newEnts.length(); ++index)
        delete newEnts[index]; // Danger Will Robinson, we’re still leaking memory
    return;
}

// Explode – bad example #3
void mySpecialExplode(const AcDbEntity* pEnt)
{
    AcDbVoidPtrArray newEnts;
    if (Acad::eOk == pEnt->explode(newEnts)) {
        // process the contents of newEnts here and add each entry to the database
    }
    for (int index = 0; index < newEnts.length(); ++index)
        // Danger Will Robinson – deleting a database resident object
        delete static_cast<AcDbObject*>(newEnts[index]);
The first example is sloppy and completely leaks the contents of newEnts.

The second example attempts to clean up after itself, but still ultimately leaks a large amount of memory because it is calling delete on a void*, so the actual destructors for the items in the array aren’t called.

The third example will result in a crash since you can’t delete a database resident object.

Here’s the correct implementation:

```cpp
// Explode – good example
void mySpecialExplode(const AcDbEntity* pEnt)
{
    AcDbVoidPtrArray newEnts;
    if (Acad::eOk == pEnt->explode(newEnts)) {
        // process the contents of newEnts here and add each entry to the database
        for (int index = 0; index < newEnts.length(); ++index) {
            AcDbObject* pObj = newEnts[index];
            if (pObj && pObj->objectId().isNull())
                delete static_cast<AcDbObject*>(pObj);
        }
    }
    return;
}
```

4. **AcDbDwgFiler::readString()**

The `AcDbDwgFiler::readString()` implementation returns an allocated string which the caller is responsible for freeing.

```cpp
// AcDbDwgFiler::readString() – bad example
Acad::ErrorStatus MyCustomObject::dwgInFields(AcDbDwgFiler* pFiler)
{
    // m_Str is a CString data member on the class
    TCHAR *name = NULL;
    if (Acad::eOk == filer->readString(&name)) {
        m_Str = name;
    }
    return Acad::eOk;
}
```

This can be rewritten one of two ways to avoid the memory leak.
// AcDbDwgFiler::readString() – good example #1
Acad::ErrorStatus MyCustomObject::dwgInFields(AcDbDwgFiler* pFiler)
{
    // m_Str is a CString data member on the class
    TCHAR *name = NULL;
    if (Acad::eOk == filer->readString(&name)) {
        m_Str = name;
        acutDelString(name);
    }
    return Acad::eOk;
}

// AcDbDwgFiler::readString() – good example #2
Acad::ErrorStatus MyCustomObject::dwgInFields(AcDbDwgFiler* pFiler)
{
    // m_Str is a CString data member on the class
    AcString name;
    if (Acad::eOk == filer->readString(&name)) {
        m_Str = name;
    }
    return Acad::eOk;
}

5. Selection sets
An ARX application can have at most 128 open selection sets. If you exceed 128 open selection sets, then all subsequent selection operations made by your application will fail to select anything until the application closes the open selection sets, or more likely when the customer gets upset that selection isn’t working and restarts the application. Because of this relatively small limit on the number of open selection sets you need to take care in your application to not leak them.

a. Pickfirst selection set
Every successful call to acedSSGetFirst() needs to be matched with a call to acedSSFree().

    // acedSSGetFirst() – bad example
    void MyCustomClass::filterPickFirstForLayer(const TCHAR* pLayerName)
    {
        struct resbuf* pickfirst = NULL;
        if (RTNORM != acedSSGetFirst(NULL, &pickfirst))
            return;

        ads_name pset;
        ads_name_set(pickfirst->resval.rlname, pset);
        int length;
        acedSSLength(pset, &length);
for (int count = 0; count < length; ++count) {
    // iterate the contents of the selection set here
}

// Danger Will Robinson! The selection set is being leaked

Here’s the proper version:

```cpp
void MyCustomClass::filterPickFirstForLayer(const TCHAR* pLayerName)
{
    struct resbuf* pickfirst = NULL;
    if (RTNORM != acedSSGetFirst(NULL, &pickfirst))
        return;
    
    ads_name pset;
    ads_name_set(pickfirst->resval.rlname, pset);
    int length;
    if (RTNORM != acedSSGetLength(pset, &length)) {
        acedSSFree(pset);
        acutRelRb(pickfirst);
        return;
    }
    
    for (int count = 0; count < length; ++count) {
        // iterate the contents of the selection set here
    }
    
    acedSSFree(pset);
    acutRelRb(pickfirst);
}
```

b. **General selection operations**

Every successful call to `acedSSGet()` must be matched with a call to `acedSSFree()`.

```cpp
// acedSSGet() – bad example
void MyCustomClass::SelectSomething()
{
    ads_name selSet;
    int ret = acedSSGet(NULL, NULL, NULL, NULL, selSet);
    if (ret != RTNORM)
        return;
    
    long selSetLength = 0;
    if (RTNORM == acedSSGetLength(selSet, &selSetLength)) {
        for (int count = 0; count < selSetLength; ++count) {
            // iterate the contents of the selection set here
        }
    }
}
```
// Danger Will Robinson! The selection set is being leaked

Here is the correct version:

// acedSSGet() – good example
void MyCustomClass::SelectSomething()
{
    ads_name selSet;
    int ret = acedSSGet(NULL, NULL, NULL, NULL, selSet);
    if (ret != RTNORM)
        return;

    long selSetLength = 0;
    if (RTNORM == acedSSLength(selSet, &selSetLength)) {
        for (int count = 0; count < selSetLength; ++count) {
            // iterate the contents of the selection set here
        }
    }
    acedSSFree(pset);
}

6. Missing virtual destructor
If you have a base class with one or more virtual methods, you need to make sure to declare the
destructor as virtual. Failure to declare a virtual destructor on a class with virtual methods will lead to
all sorts of subtle, hard to debug, issues with the code.

// Base class with missing virtual – bad example
class Base
{
    public:
        virtual void DoSomething() = 0;
        // Danger Will Robinson. Base class with virtual method but no virtual destructor
}; // end of Base definition

class MisbehavedChild : public Base
{
    public:
        MisbehavedChild ()
        {
            m_pDb = new AcDbDatabase();
        }

        virtual ~MisbehavedChild ()
With this example code, every instance of the MisbehavedChild will result in the leak of an AcDbDatabase object. A cursory examination of the code would indicate that everything appears to be working as expected since every MisbehavedChild is being assigned to an auto_ptr which will delete it when it goes out of scope, but if you set a breakpoint in the MisbehavedChild destructor you would discover that the breakpoint is never hit.

With DevStudio, there is a compiler warning you can enable to identify problems like this. In your project file you can enable compiler warning C4265 (it’s off by default) to have the compiler warn you about this type of error. The reason that this warning is off by default is that many of the ATL header files generate warnings when this setting is enabled. If you are using these headers, then you will need to do something similar to the following if you want to use this compiler setting:

```cpp
#pragma warning(push)
#pragma warning(disable:4265) // disable missing virtual destructor warning
#include <afxdao.h>       // MFC DAO database classes
#pragma warning(pop)      // C4265
```

7. **Identifying potential leaks with manual inspection**

Here are two regular expressions you can use in DevStudio to manually identify calls in your ARX code which are potentially leaking memory. The queries catch areas which we have historically found to generate a high number of leaks. As the ARX API evolves it is possible that the queries would need to be modified to capture new API implementations.
a. **Potential leaks when the supplied argument is not const**
This query identifies API calls which will result in a memory leak if the supplied argument is not declared as const (see item #2 above). The query simple identifies areas of code where the function calls are being made and a manual inspection is needed for each location to determine if a leak is actually possible.

```
[.>]+(asciiDescription|bigFontFileName|comments|fileName|getLayoutName|getName|pathName|plotStyleSheet)[ 	]*
```

b. **Potential leaks if delete is not called**
This query identifies locations in your ARX code where every call needs to be matched with a corresponding delete.

```
[.>]+(layer|contents|linetype|getDimblk|font|getProjectName|projectName|getTitle|getSubject|getAuthor|getKeywords|getComments|getLastSavedBy|getRevisionNumber|getHyperLinkBase|getCustomSummaryInfo|getLongTransactionName|tag|textString|prompt|attributeIterator)[ 	]*
```

8. **Leak detection tools**
There are numerous tools available to help identify locations in your code which are leaking memory. Two that I have personal experience with are:

a. **Bounds Checker. Part of the DevPartner offering from Micro Focus**
   [http://www.microfocus.com](http://www.microfocus.com)

b. **Memory Validator**
   [http://www.softwareverify.com](http://www.softwareverify.com)
Performance

1. **AcArray resizing**
Try to avoid repeatedly resizing the same AcArray. Each time that adding an item to the array will result in the logical length of the array exceeding the physical length, new memory is allocated and the contents of the array are copied.

   // AcArray resizing – bad example
   AcArray<AcDbEntity*> contents;
   int count = 256;
   for (int loop = 0; loop < count; ++loop)
       contents.append(new MyCustomObject); // Every 8 passes will force the array
   // to reallocate itself

   // AcArray resizing – good example
   int count = 256;
   AcArray<AcDbEntity*> contents(count); // initialize the physical length to the known ending size
   for (int loop = 0; loop < count; ++loop)
       contents.append(new MyCustomObject);

2. **Avoid needlessly opening objects for write**
When an object is opened for write, there is a lot of booking that goes on behind the scenes to ensure that stuff like the undo filer is ready to accept modifications to the object, if it is an entity being opened the layer is checked to ensure that it isn’t locked, the document lock state is checked, reactors are notified that the object is opened for write, etc. If you have no intention of modifying the object then there is no reason to open it for write and incur the expense of the additional overhead associated with opening an object for write.

   // Object open – poor example
   bool checkForClosedCurve(const AcDbObjectId& entId)
   {
       AcDbObjectPointer<AcDbCurve> pCrv(entId, AcDb::kForWrite); // Why use kForWrite?
       if (Acad::eOk != pCrv.openStatus())
           return false;
       return pCrv->isClosed();
   }

   Since this code is not trying to modify the AcDbCurve, it should not be using AcDb::kForWrite to open the object.

3. **isKindOf() and cast()**
Once you have used isKindOf() to determine the type of an object, there’s no need to use the ARX cast() method.
For example:

```cpp
// isKindOf() – bad example
if (pObj->isKindOf(AcDbLine::desc())) {
    AcDbLine* pline = AcDbLine::cast(pObj); // unnecessary performance hit
    // do something with the line pointer
}

// isKindOf() – good example
if (pObj->isKindOf(AcDbLine::desc())) {
    AcDbLine* pline = static_cast<AcDbLine*>(pObj);
    // do something with the line pointer
}
```

The reason the second form is preferred is that the isKindOf() call actually makes calls to cast(), so with the first example you are incurring the overhead making multiple cast() calls.

4. **strings vs object ids**
   
   Your ARX code should try to use an AcDbObjectId instead of a string whenever possible. For example, every layer in a drawing has an associated AcDbObjectId and an associated string for the name. Once the object id is assigned it won’t change over the life of the database, but the same isn’t necessarily true for the name string. Users can go in and rename layers. So if your application is holding onto the layer name, and then the user renames the layer, the next time your application goes to look for the layer it won’t be found. But if the application is holding onto the AcDbObjectId for the layer, then it doesn’t matter what the user renames the layer to since object ids don’t change. Also, testing two object ids for equality is quicker than performing string compares on the names.

5. **Use AcDbObject::isNull() to check for NULL object ids**
   
   Given the variable
   ```cpp
   AcDbObjectId myId;
   ```
   
   The following code samples all do the same thing
   ```cpp
   // AcDbObjectId null check example #1
   bool nullId = myId == 0;
   
   // AcDbObjectId null check example #2
   bool nullId = myId == NULL;
   
   // AcDbObjectId null check example #3
   bool nullId = myId == AcDbObjectId::kNull;
   
   // AcDbObjectId null check example #4
   ```
bool nullId = myId.isNull(); // This is the preferred method to test for a NULL object id

6. **Determine object class type without opening the object**

You can use the `AcDbObjectId::objectClass()` to determine the class type without incurring the overhead of opening the object.

```cpp
// Using AcDbObjectId::objectClass() – bad example
Acad::ErrorStatus collectAllTextFromBTR(const AcDbBlockTableRecord* pBTR,
                                       AcDbObjectIdArray& textIds)
{
  AcDbBlockTableRecordIterator *pBTRIterator = NULL;
  Acad::ErrorStatus es = pBTR->newIterator(pBTRIterator);
  if (Acad::eOk != es)
    return es;

  // Iterate over the objects in the block
  for (; !pBTRIterator->done(); pBTRIterator->step()) {
    AcDbEntity *pEnt;
    es = pBTRIterator->getEntity(pEnt, AcDb::kForRead);
    if (Acad::eOk != es)
      continue;
    if (pEnt->isKindOf(AcDbText::desc()))
      textIds.append(pEnt->objectId());
    continue; // Danger, Will Robinson!
  }
  delete pBTRIterator;
  return Acad::eOk;
}
```

The routine can be re-written more efficiently as follows:

```cpp
Acad::ErrorStatus collectAllTextFromBTR(const AcDbBlockTableRecord* pBTR,
                                       AcDbObjectIdArray& textIds)
{
  AcDbBlockTableRecordIterator *pBTRIterator = NULL;
  Acad::ErrorStatus es = pBTR->newIterator(pBTRIterator);
  if (Acad::eOk != es)
    return es;

  // Iterate over the objects in the block
  AcRxClass* pTextClass = AcDbText::desc();
  AcDbObjectId entId;
  for (; !pBTRIterator->done(); pBTRIterator->step()) {
    es = pBTRIterator->getEntityId(entId);
    if (Acad::eOk != es)
      continue;
    if (entId.objectClass()->isDerivedFrom(pTextClass))
      textIds.append(entId.objectId());
  }
  return Acad::eOk;
}
```
textIds.append(entId);
continue;
}
delete pBTRIterator;
return Acad::eOk;
}

7. **AcDbFullSubentPath::objectIds**

This method returns a reference to an AcDbObjectIdArray. If you don’t intend to modify the array the code can be rewritten to improve performance.

// AcDbFullSubentPath::objectIds() usage – bad example #1
void MyCustomClass::findPathNode(const AcDbFullSubentPath& path) {
    AcDbObjectIdArray ids = path.objectIds();
    if (ids.isEmpty())
        return
    // additional processing here
}

// AcDbFullSubentPath::objectIds() usage – bad example #2
void MyCustomClass::findPathNode(const AcDbFullSubentPath& path) {
    AcDbObjectIdArray ids;
    path.objectIds(ids);
    if (ids.isEmpty())
        return
    // additional processing here
}

Both examples can be rewritten to avoid the overhead of constructing and copying the AcDbObjectIdArray as follows:

// AcDbFullSubentPath::objectIds() usage – good example
void MyCustomClass::findPathNode(const AcDbFullSubentPath& path) {
    const AcDbObjectIdArray& ids = path.objectIds();
    if (ids.isEmpty())
        return
    // additional processing here
}
By declaring the ids variable to be a const reference, the objectIds() call results in the ids variable pointing at the same memory held by paths, so we avoid the overhead of constructing the AcArray and then copying all of the data into it.

8. **Minimize the number of AcEditorReactor classes**

ARX applications should try to use just one AcEditorReactor for the entire application. There are default implementations for every method on the class, so even if you only overload one method, the default implementations are still being called. For example, say you want to be notified when a save operation is about to start, so you implement a MyEditorReactor which derives from AcEditorReactor, and then you overload beginSave(). MyEditorReactor::beginSave() will be called every time a save begins, but for all of the other reactor methods the AcEditorReactor implementation would be called. So each time a lisp routine is started the AcEditorReactor::lispWillStart() default implementation would be called for your reactor. What typically ends up happening is that as an application evolves, one part of the code determines that they need the save notification so the app implements MyEditorReactor::beginSave(), but then somebody comes along with a new feature in a different part of the code that needs a different notification so they implement MyNewEditorReactor::lispWillStart(). Soon you end up with the app implementing multiple editor reactors, and the overall performance drops as all of the same notifications are being sent to reactors that don’t care about them.
Sample Routine

How many errors can you spot in the following code?

```cpp
// m_PickFirstIdList is defined as an AcDbObjectIdArray on a global singleton MyCustomClass
1    void MyCustomClass::filterPickFirstForLayer(const TCHAR* pLayerName)
2    {
3        struct resbuf* pickfirst;
4        acedSSGetFirst(NULL, &pickfirst);
5        ads_name pset;
6        ads_name_set(pickfirst->resval.rlname, pset);
7        int length;
8        acedSSLength(pset, &length);
9        for (int count = 0; count < length; ++count) {
10           ads_name ename;
11           AcDbObjectId entId;
12           acedSSName(pickfirst, count, ename);
13           acdbGetObjectId(entId, ename);
14           AcDbEntity* pEnt;
15           acdbOpenAcDbEntity(pEnt, entId, AcDb::kForWrite);
16           if (strcmp(pEnt->layer(), pLayerName) != 0) {
17              continue;
18           }
19           pEnt->close();
20           m_PickFirstIdList.add(endId);
21        }
22    }
```

Answer:

Line 1: Two errors.
  #1 - The routine is defined as a void, so there’s no way for the routine to return any type of error status.
  #2 – The argument is a TCHAR*. For performance this should probably be an AcDbObjectId for the layer to be searched for.

Line 3: Two errors.
  The pLayerName argument isn’t tested for NULL.
  The pickfirst variable is an uninitialized pointer.

Line 4: One error.
  The return value from acedSSGetFirst() is ignored.

Line 7: One error.
  The variable length is uninitialized.

Line 8: One error.
  The return value from acedSSLength() is ignored.

Line 12: One error.
  The return value from acedSSName() is ignored.

Line 13: One error.
The return value from acdbGetObjectld() is ignored.

Line 14: One error.
The variable pEnt is an uninitialized variable.

Line 15: Two errors.
#1 – The return value from acdbOpenObject() is ignored.
#2 – The object is being opened for write when only read access is needed.

Line 16: Two errors.
#1 – The pEnt->layer() call is leaking the returned string.
#2 – For performance reasons this should be testing object ids instead of strings

Line 17: One error:
pEnt isn’t being closed before the continue.

Line 20: One error: Crash potential.
Since m_PickFirstIdList is a member of a global singleton, if the id isn’t removed from the array
prior to the current database being destroyed, the next attempt to use the array will result in a

crash.

Line 22: Two errors.
#1 – The selection set is being leaked.
#2 – The resbuf pickfirst is being leaked.

That’s 18 errors in 22 lines of code.

Q & A